# Transparent Pointer Compression for Linked Data Structures

Chris Lattner Vikram S. Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu

## ABSTRACT

64-bit address spaces are increasingly important for modern applications, but they come at a price: pointers use twice as much memory, reducing the effective cache capacity and memory bandwidth of the system (compared to 32-bit address spaces). This paper presents a sophisticated, automatic transformation that shrinks pointers from 64-bits to 32-bits. The approach is "macroscopic," i.e., it operates on an entire logical data structure in the program at a time. It allows an *individual* data structure instance or even a subset thereof to grow up to $2^{32}$ bytes in size, and can compress pointers to some data structures but not others. Together, these properties allow efficient usage of a large (64-bit) address space. We also describe (but have not implemented) a dynamic version of the technique that can transparently expand the pointers in an individual data structure if it exceeds the 4GB limit. For a collection of pointer-intensive benchmarks, we show that the transformation reduces peak heap sizes substantially by (20% to 2x) for several of these benchmarks, and improves overall performance significantly in some cases.

## Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers, Optimization, Memory management

## General Terms

Algorithms, Performance, Experimentation, Languages

## Keywords

Recursive data structure, data layout, cache, static analysis, pointer compression

## 1. INTRODUCTION

64-bit computing is becoming increasingly important for modern applications. Large virtual address spaces are important for several reasons, including increasing physical memory capacity, rapidly growing data sets, and several advanced programming techniques [15, 18, 6].

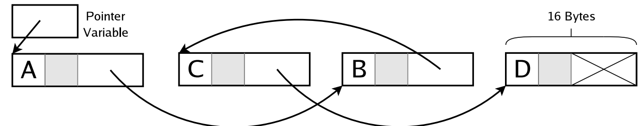



**Figure 1: Linked List of 4-byte values**

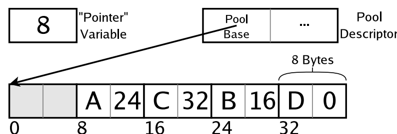



**Figure 2: Pointer Compressed Linked List**

One problem with 64-bit address spaces is that 64-bit pointers can significantly reduce memory system performance [12] compared to 32-bit pointers. In particular, pointer-intensive programs on a 64-bit system will suffer from (effectively) reduced cache/TLB capacity and memory bandwidth for the system, compared to an otherwise identical 32-bit system. The increasing popularity of object oriented programming (which tends to be pointer intensive) amplify the potential problem. We observe that the primary use of pointers in many programs is to traverse linked data structures, and very few *individual* data structures use more than 4GB of memory, even on a 64-bit system. The question therefore is how can we use pointers more efficiently to index into individual data structures?

This paper presents a sophisticated compiler transformation, Automatic Pointer Compression, which transparently converts pointers to use a smaller representation (e.g. from 64-bits to 32-bits) for a subset of the data structures in a program. Pointer Compression builds on a previously published technique called Automatic Pool Allocation [10] to modify the program to allocate and free objects from pools of memory within the system heap. Automatic Pool Allocation attempts to segregate heap objects from distinct data structures into separate pools. It guarantees that every static pointer variable or pointer field pointing into the heap is mapped to a unique pool descriptor at compile time. Pointer Compression compresses a 64-bit pointer by replacing it with a smaller integer index from the start of the corresponding pool.

Consider a simple linked list of integers. Figure 1 illustrates the list compiled without pointer compression, and Figure 2 illustrates the memory organization with pointers compressed to 32-bit integer indexes. In this example, each

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*MSP'05,* Chicago, USA.
Copyright 2005 ACM 1-59593-147-3/05/06 ...$5.00.

node of the list originally required 16 bytes of memory[1] (4 bytes for the integer, 4 bytes of alignment padding, and 8 bytes for the pointer), and the nodes may be scattered throughout the heap. In this (extreme) example, pointer compression reduces each node to 8 bytes of memory (4 for the integer, and 4 for the index that replaces the pointer). Each index holds the byte offset of the target node from the start of the pool instead of an absolute address in memory.

We describe and evaluate a "static" version of pointer compression that limits individual pools to $2^k$ bytes each, for some $k < 64$ (e.g., $k = 32$), fixed at compile-time for each pool. It can be applied selectively, i.e., other pools can grow to the full $2^{64}$ bytes. We show that this transformation provides substantial reductions in memory consumption and, in some cases, significant net performance improvements over pool allocation alone (even though pool allocation itself has already improved memory hierarchy performance substantially in many cases).

We also describe an optional "dynamic" version of the transformation that can expand indices for a particular pool transparently at run-time from $k$ to 64 bits when a pool exceeds $2^k$ bytes. This transformation ensures that the technique is fully transparent, but is restricted in applicability to type-safe data structures where pointers do not point into the middle of objects.

We begin by describing two underlying techniques - a pointer analysis (called Data Structure Analysis) and Automatic Pool Allocation - that provide the foundation for Pointer Compression. We then describe the static and dynamic versions of the technique, several simple optimizations that can improve its performance, and finally experimental results and related work.

## 2. BACKGROUND INFORMATION

The broad goal of Automatic Pool Allocation is to enable *macroscopic* compiler transformations by giving the compiler information and control over data structure layouts. Pointer Compression is one such client. Both Pool Allocation and Pointer Compression operate on a common points-to graph representation with specific properties, which we refer to as **DS Graphs**. Below, we describe the DS Graph representation and then briefly summarize the Pool Allocation transformation. The precision (but not the correctness) of both transformations is affected by how DS graphs are computed. Therefore, we also very briefly describe relevant aspects of the pointer analysis we use to compute DS graphs, which we call Data Structure Analysis (DSA) [7].

Figure 4 shows a simple linked-list example and the DS graphs computed by DSA for the three functions in the example. We will use this as a running example in this Section and the next.

### 2.1 Points-to Graph Representation: DS Graphs

The key properties of the points-to graph representation (DS graphs) required for this work are as follows. Like any points-to graph, a DS graph is a directed graph that provides a compile-time representation of the memory objects in a program and the points-to relationships between them. Each node within a DS graph represents a distinct set of memory objects. The current work assumes four key properties for the graph:

[1] Not including the header added by malloc, typically 4 bytes.

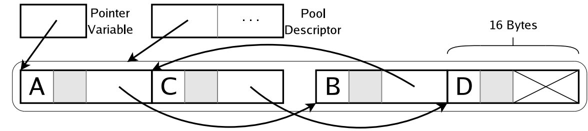



**Figure 3: Pool-allocated Linked List**

1. *Type information*: Each node, $n$, is associated with a type, $n.\tau$, that is some program-defined type, or $\perp$ representing an unknown type. Any DS node with $\tau \neq \perp$ represents a set of objects of a single type, $\tau$, i.e., the compiler has proven that that all operations on pointers to the node are consistent with $\tau$. We refer to such nodes as type-homogeneous (TH) nodes. If $\tau = \perp$, this is treated like an unknown-size array of bytes and is shown as $\tau =$ `byte` plus an **A**(rray) flag in the figures. Figure 4 shows that DSA constructs a TH node for the objects of type $\tau =$ `list` in functions `MakeList` and `Length` and for the $A$ list in function `Testlists`. It marks the $B$ list as non-TH because a location in a $B$ list object is accessed as a character.

2. *Memory classes*: Memory objects are distinguished into four classes: Heap, Stack, Global and Unknown. For each node, this is represented as a set of flags, $M \subseteq \{\mathbf{H}, \mathbf{S}, \mathbf{G}, \mathbf{U}\}$. A node with $\mathbf{U} \in M$ must be assigned type $\tau = \perp$. Functions are treated simply as Global objects so that function pointers are represented uniformly in the graph.

3. *Field-sensitive points-to information*: An edge in a DS graph is a 4-tuple $\{s, f_s, t, f_t\}$. $s$ and $t$ are DS nodes, while $f_s$ and $f_t$ are field numbers of $s.\tau$ and $t.\tau$ respectively. Only structure types have multiple fields, i.e., scalar or array types are treated as a single field.

4. *Single target for each pointer*: Every pointer variable or field has a single outgoing edge, i.e., all objects pointed to by a common pointer must be represented by a single node. The benefits of this property for the two transformations are explained in the next subsection.

In computing DS graphs as defined above, the naming scheme used to distinguish heap objects can have a strong influence on the outcome of transformations such as Pool Allocation and Pointer Compression, both of which focus on heap objects. In particular, in order to segregate distinct data structures into separate pools (and apply pointer-compression to each data structure separately), the pointer analysis must use a *fully context-sensitive* naming scheme for heap objects, i.e., distinguish heap objects by acyclic call paths. This is necessary in order to segregate two instances of a data structure (e.g., two distinct binary trees) created and processed by common functions. Both transformations would work with a less precise naming scheme, e.g., one based on allocation sites, but would not segregate objects from two such data structures.

We have developed a very fast context-sensitive pointer analysis algorithm we call "Data Structure Analysis" (DSA) [7] to compute the DS graphs used by pool allocation and pointer compression. DSA analyzes programs of 100K-200K lines of code in 1-3 seconds, and takes a small fraction of the time (about 5% or less) taken by `gcc -O3` to compile the same programs. DSA distinguishes heap objects by entire acyclic call paths, as described above. For example, in function `Testlists`, this property enables DSA to distinguish

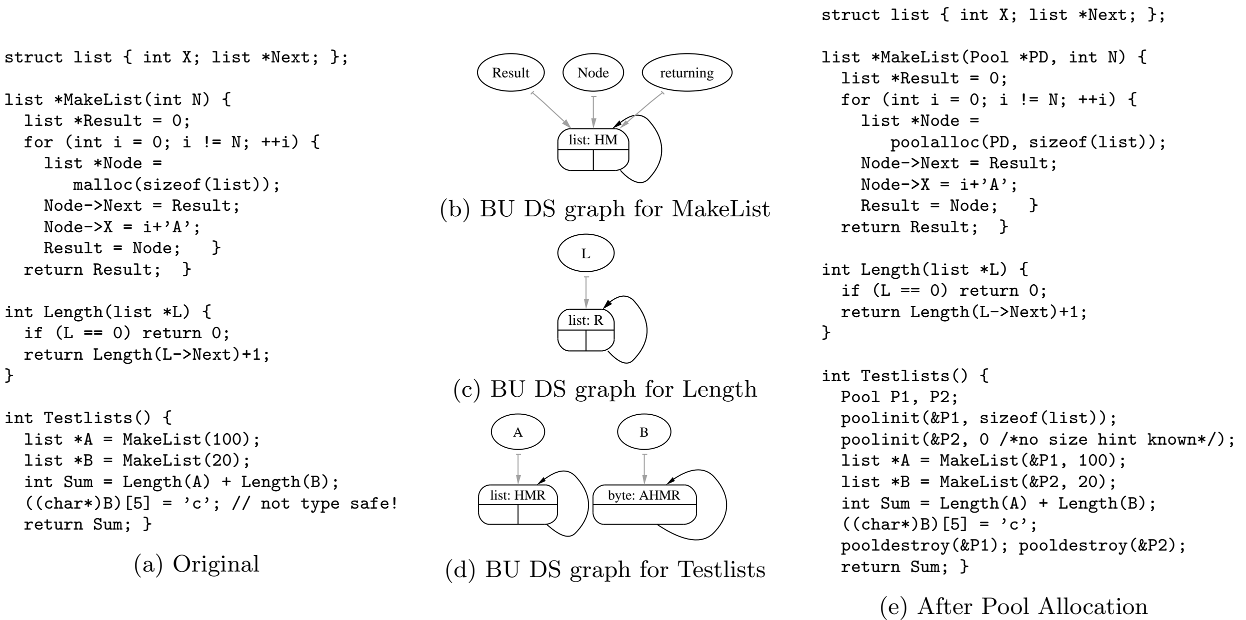



**Figure 4: Simple linked list example**

the objects in the lists $A$ and $B$ (and prove the lists are disjoint), even though they are created and manipulated by common functions. If heap objects were instead named by allocation site, $A$ and $B$ would point to a single node in the graph and objects of the two lists would not be distinguishable.

A second feature of DSA is that, like many other context-sensitive pointer analyses, e.g., [5, 4, 11, 13], it actually computes two points-to graphs for each function in a program - a bottom-up (BU) graph representing a function and its callees (but not any callers), and a final top-down (TD) graph representing the effects of both callees and callers. The TD graph is the final result of the pointer analysis. The BU graph, however, provides a more precise basis for both Pool Allocation and Pointer Compression because two distinct pointer arguments in a function may be aliased (point to the same DS graph node) in one calling context but not in another. Using the BU graph allows the transformations to distinguish (and therefore) segregate objects in the latter case. Therefore, pointer compression operates largely using the BU graph, except where noted, and pool allocation only uses the BU graph [10].

The example graphs also illustrate other basic but relevant features of DSA. The algorithm detects that the structure is recursive (the cycle in the graph), and that the pointers in the functions point to the list objects. In `MakeList`, DSA also detects that heap objects are allocated (**H**) and returned, whereas in `Length`, memory is not allocated or freed (no **H** flag). (The **M** and **R** flags shown in the figures can be ignored for this work.)

## 2.2 Automatic Pool Allocation

Given a program with calls to `malloc` and `free`, Automatic Pool Allocation modifies the program to allocate and free memory from pools within the standard system heap. A pool is represented in the program by a *pool descriptor*. Automatic Pool Allocation creates one distinct pool descriptor for each node marked **H** in a function's DS graph, effectively partitioning objects in the heap as they were partitioned in the DS graph by pointer analysis. Calls to `malloc` and `free` are simply replaced with calls to `poolalloc` and `poolfree`, passing in the pool descriptor corresponding to the DS node pointed to by the relevant pointer. This implies that the lifetime of individual objects allocated from the pool stay exactly the same as the original program (except for some leaked objects as explained below).

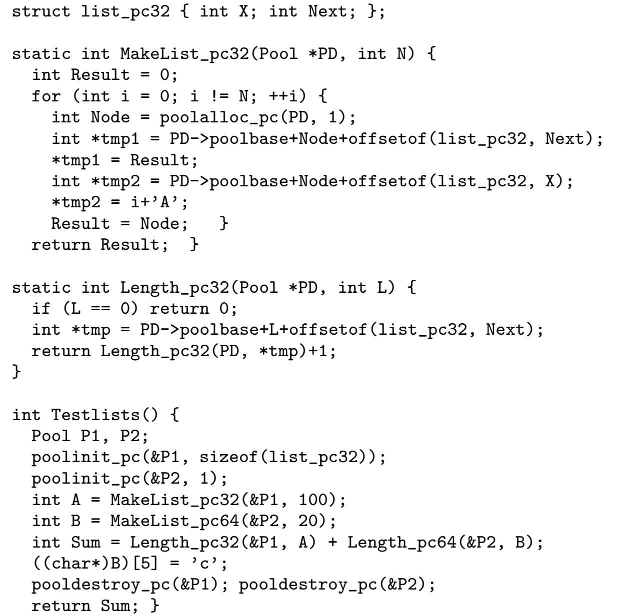



**Figure 5: Example after static compression**

The pool runtime library provides functions `poolinit` and `pooldestroy` to initialize and destroy a pool descriptor. `pooldestroy` also releases any remaining pool memory to the system heap. (Note that this can reclaim memory for objects that were never freed in the original program, i.e., were previously leaked.) Each pool is a fully general heap, providing equivalents for all standard heap functions, including `poolalloc`, `poolfree`, `poolrealloc` and `poolmemalign`. A pool internally obtains and frees memory off the system heap in large slabs using `malloc` and `free`.

Figure 3 shows the result of pool-allocating the linked list in Figure 1. The linked list has been placed into one memory pool, containing two slabs of memory, each holding two nodes (in practice, many more nodes would fit in each slab).

Pool allocation also uses the DS graph to identify which memory objects do not escape from a function (e.g. the "A" and "B" lists in `Testlists`). The pools for these DS nodes are created and destroyed at the entry and exits of the function. For nodes pointed to by a global variable, no such function will be found and the pool create/destroy calls are inserted into `main`. (Note that this only affects the lifetime of the pool, and not the individual objects within the pool.)

Figure 4(e) shows the example code transformed by pool allocation. Because the `MakeList` function allocates heap memory and returns it (thus escaping from the function), a pool descriptor argument is added to the function indicating the pool of memory to allocate from. If a function calls any other functions that require pool arguments (e.g. as `Testlists` calls `MakeList`), pool descriptors are passed into the function call. The `length` function is unmodified by pool allocation because it does not allocate or free any memory (nor do any callees). Note that pool allocation does not require type-safety, allowing it to pool allocate the "B" list.

Automatic Pool Allocation, combined with DSA, provides some key properties for this work:

- Because of the context-sensitivity of DSA, Automatic Pool Allocation will segregate objects from distinct data structures (identified by DSA) into different pools. For example, lists $A$ and $B$ are allocated out of distinct pools in Figure 4(e).

- Because of the "single-target" property of DS graphs, Automatic Pool Allocation guarantees that every variable or field pointing into the heap points to a single pool, known at compile time. This ensures that the mapping between a pointer and a pool descriptor does not have to be tracked dynamically, which can be complex and inefficient for pointers stored in memory.

  Below, we use $Node(p)$ for the DS node corresponding to pointer expression $p$, $Pool(n)$ for the pool descriptor corresponding to DS node $n$, and $Pool(p)$ as an abbreviation for $Pool(Node(p))$.

- A TH node (with $\tau \neq \perp$) in the DS graph generates a pool where all objects are of type $\tau$ or (if the **A** flag is set) arrays of $\tau$. Every individual $\tau$ item is identically aligned, i.e., the compiler knows the exact layout of items relative to the pool base.

# 3. STATIC POINTER COMPRESSION

Static pointer compression reduces the size of pointers in data structures in two steps. First, it replaces pointers in data structures with integers representing offsets from a pool base (i.e., indexes into the pool). Second, in order to compress this index, it attempts to select an integer type that is smaller than the pointer size (e.g. by using a 32-bit integer on a 64-bit host). We refer to these as "index conversion" and "index compression" respectively. The latter step may fail because it requires somewhat stronger safety guarantees[2].

[2]Note that index conversion alone may also be useful for purposes other than pointer compression because it provides "position independent" data structures that can be relocated in memory *without rewriting any pointers* other than the pool base.

Static pointer compression will cause a runtime error if the program allocates more than $2^k$ bytes from a single pool using $k$-bit indices. Techniques to deal with this in the static case are discussed briefly in Section 3.5. Alternatively, this problem is solved by the dynamic algorithm in Section 4, but that algorithm is more restrictive in its applicability.

For our list example of Figure 4(a), the static pointer compression transformation transforms the code to that in Figure 5. Both the `A` and `B` lists are index-converted. Indices in the `A` list to itself are compressed whereas those in the `B` list to itself are not, for reasons explained below. This also requires that distinct function bodies be used for the `A` and `B` lists (those for the former are shown). By shrinking pointers from 64-bits to 32-bits (which also reduces intra-object padding for alignment constraints), each object of the `A` list is reduced from 16 to 8 bytes – effectively reducing the cache footprint and bandwidth requirement by half for these nodes. The dynamic memory layout of the `A` list is transformed from that of Figure 3 to Figure 2.

Below, we first describe changes required to the pool allocation runtime to support pointer compression. We then describe the transformation in three stages: the legality criteria, the transformation for data structures that are never passed to or returned from functions, and finally the approach to handle function calls.

## 3.1 Pointer Compression Runtime Library

The pointer compression runtime library is almost identical to the standard pool allocator runtime described briefly in Section 2.2 and in more detail in [10]. The only two functionality differences are that it guarantees that the pool is always contiguous (discussed in Section 5.1) and that it reserves the $0^{th}$ node to represent the null pointer. The library interface is also cosmetically different in that the memory allocation/free functions take indices instead of pointers. The API is listed Figure 6, below.

`void poolinit_pc(Pool* PP, unsigned NodeSize);`
Initialize the pool descriptor; record node size
`void pooldestroy_pc(Pool* PP)`
Release pool memory and destroy pool descriptor
`int poolalloc_pc(Pool* PP, uint NumBytes)`
Allocate `NumBytes` bytes.
`void poolfree_pc(Pool* PP, int NodeIdx)`
Mark the object identified by `NodeIdx` as free.
`void* poolrealloc_pc(Pool* PP, int NodeIdx ptr, uint NumNodes)`
Resize an object to `NumNodes` nodes.

**Figure 6: Pool Compression Runtime Library**

## 3.2 Safety Conditions for Static Pointer Compression

The two steps of pointer compression (index conversion and index compression) have separate legality criteria. A key point to note is that these two sets of criteria apply to *potentially different pools*. For example, consider the points-to graph in Figure 12(a). Compressing the pointers from `list2` to the `int` objects requires (i) making the `int` pool an indexed pool (i.e., using indices instead of pointers to objects in this pool), but (ii) compressing the indices stored in the `list2` pool. Therefore, the legality criteria for index conversion apply to the `int` pool and for index compression

apply to the `list2` pool. We refer to these as the *indexed pool* and the *source pool*, respectively. Of course, for pointers within recursive data structures (e.g., the pointer from `list2` to itself), both pools are the same.

A common criterion for both index conversion and index compression is that pointers to a pool must not escape from the available program to unavailable functions such as an external library. This criterion is already guaranteed by pool allocation because the compiler cannot pool-allocate objects that escape.

*Index Conversion*: A pool is safe to access via indexes instead of pointers if the DS node corresponding to the pool represents only heap objects and no other class of memory (Global, Stack or Unknown). This condition can be determined directly from the memory flags in the DS node. It is required because stack and global data are not allocated out of a heap pool, and pointers to such objects cannot easily be converted into offsets relative to the base of such a pool.

Each indexable pool identified by this criterion will be used to hold at most $2^k$ bytes, $k < n$, where $n$ is the pointer size for the target architecture (e.g., $k = 32$ and $n = 64$). *All* valid pointers into such pools are replaced with indexes at compile time (by transforming all instructions that use these pointers, as described in the next two subsections). Only some of these indexes, however, can be compressed to use a $k < n$ bit representation; the others must still use a full $n$ bit representation (i.e., 0-extended from $k$ to $n$ bits).

*Index Compression*: An index variable contained in an object is safe to compress if the compiler can safely change the layout of the enclosing object. This is possible for objects represented by TH nodes in the points-to graph since all operations on such objects use a known, consistent program type, $\tau$ (see Section 2.1). Index values in such objects are stored using $k$-bits. For example, in Figure 4, the $A$ list objects can be reorganized and therefore can hold compressed indices whereas the $B$ list objects cannot (this would still be true if both lists pointed to a common indexed pool). Note that this criterion applies to DS nodes representing heap, global or stack objects, i.e., indexes in any of these locations can be compressed when the criterion is met.

It is important to note that type homogeneity in a pool is sufficient *but not necessary* for changing the layout of objects in the pool. In particular, if objects in a pool are of multiple types but all objects are provably accessed in a type-safe manner (e.g., when using a type-safe language), then index compression would be safe. This is particularly important for object oriented languages because, in such languages, it may be common for pools to contain objects of multiple different types derived from a common base type.

## 3.3 Intraprocedural Pointer Compression

Given the points-to graph and the results of automatic pool allocation, intraprocedural static pointer compression is relatively straight-forward. The high level algorithm is shown in Figure 7. Each function in the program is inspected for pools created by the pool allocator. (Pools passed in via arguments or passed to or received from callees are ignored for now, and considered in the next subsection.) If index-conversion is safe for such a pool, any instructions in the function that use a pointer to objects in that pool are rewritten to use indexes off the pool base. These indexes are stored in memory in compressed form ($k$ bits) when safe, otherwise left in uncompressed form (i.e., 0-extended to 64 bits). The pool is also marked to limit its aggregate size to $2^k$ bytes.

**pointercompress**(program $P$)
1 poolallocate($P$) // First, run pool allocator
2 $\forall F \in$ functions($P$)
3 set $PoolsToIndex = \emptyset$
4 $\forall p \in$ pools($F$) // Find all pools
5 if ($safetoindex(p)$) // index-conversion safe for $p$?
6 $PoolsToIndex = PoolsToIndex \cup \{p\}$
7 if ($PoolsToIndex \neq \emptyset$)
8 rewritefunction($F$, $PoolsToIndex$)

**Figure 7: Pseudo-code for pointer compression**

Once the indexable pools and the compressible index variables have been identified in the function, a single linear scan over the function is used to rewrite instructions that address the indexable pools. Assuming a simple C-like representation of the code which has been lowered to individual operations, the rewrite rules are shown in Figure 8. Operations not shown here are unmodified. In particular, any operations that cast pointers to and from integers or perform arithmetic on integers do not need to be modified for the transformation.

| Original Statement | | Transformed Statement |
|---|---|---|
| $P$ = null | $\Rightarrow$ | $P' = 0$ |
| $P_1 = P_2$ | $\Rightarrow$ | $P'_1 = P'_2$ |
| cc = $P_1 <> P_2$ | $\Rightarrow$ | cc = $P'_1 <> P'_2$, $<> \in \{<, \leq, >, \geq, ==\}$ |
| $P_1 = \&P_2\text{->}field$ | $\Rightarrow$ | $P'_1 = P'_2$ + newoffsetof($field$) |
| $P_1 = \&P_2[V]$ | $\Rightarrow$ | $P'_1 = P'_2$ + V*newsizeof($P_2[0]$) |
| *If node($P$) is non-TH or $\tau$ not a pointer ($P : \tau*$):* | | |
| V = *($\tau$*)$P$ | $\Rightarrow$ | Base = PD->PoolBase<br>V = *($\tau$*)(Base+$P'$) |
| *(($\tau$*)$P$) = V | $\Rightarrow$ | Base = PD->PoolBase<br>*($\tau$*)(Base+$P'$) = V |
| *If node($P$) is TH and $\tau$ is a pointer ($P : \tau*$):* | | |
| $P_1$ = *$P$ | $\Rightarrow$ | Base = PD->PoolBase<br>$P'_1$ = *($IdxType$*)(Base+$P'$) |
| *$P_1$ = $P$ | $\Rightarrow$ | Base = PD->PoolBase<br>*($IdxType$*)(Base+$P'_1$) = $P'$ |
| $P$ = poolalloc(PD, N) | $\Rightarrow$ | Tmp = (N/OldSize)*NewSize<br>$P'$ = poolalloc_pc(PD, Tmp) |
| poolfree(PD, $P$) | $\Rightarrow$ | poolfree_pc(PD, $P'$) |
| poolinit(PD, Size) | $\Rightarrow$ | Tmp = (Size/OldSize)*NewSize<br>poolinit_pc(PD, Tmp) |
| pooldestroy(PD) | $\Rightarrow$ | pooldestroy_pc(PD) |

**Figure 8: Rewrite rules for pointer compression**

In the rewrite rules, $P$ and $P'$ denote an original pointer and a compressed index. $V$ is any non-compressed value in the program (a non-pointer value, a non-converted pointer, or an uncompressed index). $IdxType$ is the integer type used for compressed pointers (e.g. int32_t if $k = 32$). All $P'$ values are of type $IdxType$. Indexes loaded from (or stored to) non-TH pools are left in their original size whereas those from TH pools are cast to $IdxType$.

The rules to rewrite addressing of structures and arrays lower addressing to explicit arithmetic, and use new offsets and sizes for the compressed objects, not the original. Memory allocations scale (at runtime) the allocated size from the old to the new size. The most common argument to a `poolalloc` call is a constant that is exactly "OldSize", allowing the arithmetic to constant fold to NewSize. The dynamic instructions are only needed when allocating an array of elements from a single `poolalloc` site, or when a malloc wrapper is used (in the interprocedural case).

## 3.4 Interprocedural Pointer Compression

Extending pointer compression to support function calls and returns requires three changes to the algorithm above.

First, a minor change is needed to the pool allocation transformation to pass pool descriptors for all pools accessed in a callee (or it's callees), not just those pools used for `malloc` or `free` in the callee [10]. In Figure 4 for example, the `Length` function now gets a pool descriptor argument for "`L`." Second, the rewrite rules in Figure 9 must be used to rewrite function calls and returns. These rules simply pass or return a compressed (i.e., $k$-bit) index value for every pointer argument or return value pointing to an indexable pool.

| Original Statement | | Transformed Statement |
|---|---|---|
| $P_1 = F(P_2, \mathrm{V}, P_3, ...)$ | $\Rightarrow$ | $P_1' = F_c(P_2', \mathrm{V}, P_3', ...)$ |
| $\mathrm{V}_1 = F(\mathrm{V}_2, P_2, ...)$ | $\Rightarrow$ | $\mathrm{V}_1 = F_c(\mathrm{V}_2, P_2', ...)$ |
| $F(\mathrm{V}_1, \mathrm{V}_2, ...)$ | $\Rightarrow$ | $F(\mathrm{V}_1, \mathrm{V}_2, ...)$ |
| return $P$ | $\Rightarrow$ | return $P'$ |

**Figure 9: Interprocedural rewrite rules.**
*Pool descriptor args. added by pool allocation are not shown. They are ignored during pointer compression.*

Third, and most significantly, interprocedural pointer compression must handle the problem that a reference in a function may use either compressed or non-compressed indices in different calling contexts. This problem arises because the same points-to graph node in a callee function can correspond to different pools in different calling contexts. One context may pass a TH pool and another a non-TH pool, requiring different code to load or store pointers in these two pools. We propose two possible solutions to this problem. The first is to generate conditional code for loads and stores of such index values (*uses* of these indexes are not a concern because they are always used as $n$-bit values). The second is to use function cloning and generate efficient, unconditional code in each function body. As explained in the next section, dynamic pointer compression *requires* conditional code sequences in any case to handle dynamic pool expansion, and we describe the former solution there. Our goal with static pointer compression is to present a very efficient solution that works in most common cases, and therefore we focus on the latter solution (function cloning) here. In practice, we believe that relatively little cloning would be needed for many programs.

Figure 4 shows a case when cloning must be used. In particular, `Testlists` in Figure 4 calls `MakeList` and `Length` and passes or gets back data from indexed pools into each of them. Since the $A$ list indices are compressed but the $B$ list ones are not, the transformation needs to create two versions of `MakeList` and `Length`, one for each case. The $A$ list version (denoted by suffix "_pc32") is shown; the second version is the same except it uses the uncompressed rewrite rules for loads and stores of pointers in Figure 9. Only two versions are needed for each function because only one pool within each function (the `list` node) is accessed in multiple ways. In the worst case, cloning can create an exponential number of clones for a function: one clone for each combination of compressed or uncompressed pools passed to a function. In practice, however, we find that we rarely encounter cases where TH and non-TH pools containing heap objects point to a common indexed pool or are passed to the same function.

Given the extensions described above, interprocedural static pointer compression is a top-down traversal of the program call graph, starting in main and cloning or rewriting existing function bodies as needed. All together, applied to the example in Figure 4, static pointer compression produces the code in Figure 5.

Our implementation of static pointer compression does not support indirect function calls, so a single callee occurs at each call site. However, transforming indirect function calls and their potential callees should not be technically difficult. This is because Automatic Pool Allocation already merges the DS graphs of all functions at any indirect call site into a single graph so as to pass identical pool arguments to all such functions via the call. By using this merged graph, all potential callees and the call site will automatically be consistently transformed.

## 3.5 Minimizing Pool Size Violations with Static Compression

Static compression is not a completely safe transformation because a correct program may fail if it tries to allocate more than $2^k$ bytes from a pool that uses $k$-bit indices. Nevertheless, we believe this transformation can be used safely in practice on many programs. First, each pool only holds a single instance of a data structure or even a subset of an instance (if the data structure consists of multiple nodes in the points-to graph). This means that part or all of a *single* DS instance must exceed $2^k$ bytes (e.g., 4GB for $k$=32) before an error occurs.

Second, many pools can be indexed by *objects* instead of *bytes*, thus expanding the effective maximum pool size greatly[3]. Node indexing is safe to use for TH pools holding objects for which the address of a field is not taken (i.e., all pointers point to the start of pool objects). This criterion is met by many objects in C and C++ programs, and all those in Java programs.

Third, a compiler could use profiling runs (and simple runtime pool statistics) to identify pool instances that grow unusually large compared with other pools in a program and simply disable pointer compression for those pools. Finally, programmers could use options or #pragmas to specify that pools created in certain functions should not undergo index compression.

# 4. DYNAMIC POINTER COMPRESSION

Dynamic pointer compression aims to allow an indexed pool to grow beyond the limit of $2^k$ bytes (or $2^k$ objects) by expanding compressed indices in source pools transparently at run time. Compared with static compression, this technique has a higher runtime overhead and may require more pointers not to be compressed in C and C++ programs (this is not a problem in Java programs).

There are several possible ways to implement dynamic pointer compression. To make it as simple as possible to grow pools at run time, we impose three restrictions on the optimization. First, we allow only two possible index sizes to be used for a pool: the initial $k$ bits (e.g., 32) and the original pointer size, $n$ (e.g,. 64).

Second, we compress and dynamically expand indices within objects in a source pool only if it *is also an indexed pool, and it meets the criteria for object-indexing mentioned above*: it must be a TH pool and the address of a field is not taken. This is important because expanding objects in a pool does not change their object index, although it does

[3]Object-indexing is actually required for dynamic compression, and is described below.

change their byte offset (the specific run-time operations are described below). Therefore, when objects in a source pool are expanded (and the source pool itself is indexed), the node index values in the source pool do not need to change, only their sizes increase.

Third, for any object containing compressible indices, we allow only two choices: all indices are compressed or all are uncompressed. For example, in list *list*1 in Figure 12, either both index fields (the edge to *list*2 and the back edge to the *list*1) are stored in compressed form or both are stored in uncompressed form. Note that because our pools mirror the static DS graph, a particular pointer field in all objects in a pool will point to objects in some common pool. This property plus the third restriction together imply that a single boolean flag *per pool* is sufficient to track whether indices in objects in the pool are compressed or not.

Section 4.1 describes the modified rewrite rules for dynamic pointer compression, Section 4.2 describes changes to the runtime, and Section 4.3 describes the needed changes to the interprocedural transformation.

## 4.1 Intraprocedural Dynamic Compression

Intraprocedural dynamic pointer compression is largely the same as static compression but more complex load/store code sequences are needed for objects containing compressed indices, since these indices may grow at run time. In each source pool, we store a boolean value, "*isComp*," which is set to true when objects in the pool hold compressed indices and false otherwise. If a source pool is also an indexed pool, all index values pointing to the pool held in registers, globals, or stack locations are represented using the full $n$ bits (the high bits are zero when $isComp = true$). Without this simplification, pointer compression would have to expand the indices in all such objects when the pool exceeded $2^k$ nodes. This is technically feasible for global and stack locations (using information from the points-to graph) but probably isn't worth the added implementation complexity.

Figure 11 shows the main[4] rewrite rules used for dynamic pointer compression. The transformed version of the `Length` function in the example is shown in Figure 10. Because we do not compress pool indexes if the address of a field is taken, the code for addressing the field and loading it is handled by one rule. The generated code differs from the static compression case in two ways: 1) both compressed and expanded cases must be handled; and 2) object-indexing rather than byte-indexing is used, i.e., the pool index is scaled by the node size before adding to `PoolBase`. The object size and field offsets are fixed and known at compile-time for each case.

```
/* Length with dynamic pointer compression (64->32 bits) */
static int Length(Pool *PD, long L) {
  if (L == 0) return 0;
  long Next=(PD->isComp)? (long)*( int*)(PD->PoolBase + L*8  + 4)
                        :        *(long*)(PD->PoolBase + L*16 + 8);
  return Length(PD, Next)+1;
}
```

**Figure 10: Example after dynamic compression**

The rewrite rules use a branch sequence even for *non-pointer values* ($V$). By recording the object size in the pool, however, we can use a branch-free sequence for loads and stores of non-pointer values, and can tune the sequence for the possible object sizes since these are known at compile time. For example, if a load of `L->X` occured in the `Length()` function above, it could be translated to:

```
int X = *(int*) (PD->PoolBase +
                 L << PD->LogObjSize + 0);
```

where `PD->LogObjSize` is either 3 or 4 corresponding to the compressed and uncompressed cases. (If the size is not an exact power of 2, a sequence of shifts and adds would be needed, but the code sequence can still be generated at compile-time.) Furthermore, since the pool is TH, we can safely reorder fields within the pool data type, $\tau$, *so that all non-pointer fields occur before any pointer fields.* This would ensure that the field offset for every non-pointer field is a fixed compile-time constant (e.g., the offset was 0 for `L->X` above).

| Original Statement | | Transformed Statement |
|---|---|---|
| $P$ = null | $\Rightarrow$ | $P'$ = 0 |
| $P_1 = P_2$ | $\Rightarrow$ | $P'_1 = P'_2$ |
| cc = $P_1 \stackrel{?}{=} P_2$ | $\Rightarrow$ | cc = $P'_1 \stackrel{?}{=} P'_2$ |
| $P_1 = P_2$->$field$ | $\Rightarrow$ | char *Ptr = PD->PoolBase<br>if (PD->isComp) {<br>Ptr += $P'_2$*newsizeof($pooltype$)<br>Ptr += newoffsetof($field$)<br>$P'_1$ = *(int32_t*)Ptr<br>} else {<br>Ptr += $P'_2$*oldsizeof($pooltype$)<br>Ptr += oldoffsetof($field$)<br>$P'_1$ = *(int64_t*)Ptr<br>} |
| $\tau V$ = $P$->$field$ | $\Rightarrow$ | char *Ptr = PD->PoolBase<br>if (PD->isComp) {<br>Ptr += $P'$*newsizeof($pooltype$)<br>Ptr += newoffsetof($field$)<br>} else {<br>Ptr += $P'$*oldsizeof($pooltype$)<br>Ptr += oldoffsetof($field$)<br>}<br>$V$ = *($\tau$*)Ptr |
| $P$ = poolalloc(PD, N) | $\Rightarrow$ | Tmp = N/OldSize<br>$P'$ = poolalloc_pc(PD, Tmp) |
| poolfree(PD, $P$) | $\Rightarrow$ | poolfree_pc(PD, $P'$) |
| poolinit(PD, Size) | $\Rightarrow$ | poolinit_pc(PD, &TypeDesc,<br>$PD_1$, $PD_2$, ..., NULL) |
| pooldestroy(PD) | $\Rightarrow$ | pooldestroy_pc(PD) |

**Figure 11: Dynamic pointer compression rules**

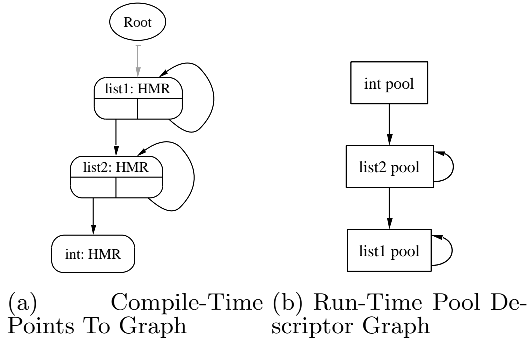
## 4.2 Dynamic Compression Runtime Library

The dynamic pointer compression runtime library is significantly different from the library for the static case. When a pool $P$ grows beyond the $2^k$ limit, the run-time must be able to find and expand (0-extend) all indices in all source pools pointing to this pool. This requires knowing which source pools point to pool $P$, and where the pointers lie within objects in these source pools.

To support these operations, the `poolinit` function takes static information about the program type for each pool (this type is unique since we only operate on TH pools), and is enhanced to build a *run time pool points-from graph* for the program. The type information for a pool consists of the type size and the offset of each pointer field in the type.

The run time pool points-from graph has a node for each pool and an edge $P_2 \rightarrow P_1$, if there is an edge $N_1 \rightarrow N_2$ in the compiler's points-to graph, where $P_1$ and $P_2$ are the pools for nodes $N_1$ and $N_2$. An example points-to graph and the run-time points-from graph are shown in Figure 12(b). When `poolinit_pc` is called to initialize a

[4] We only show the rules for loads of structure fields. Stores are identical except for the final instruction, and array accesses are similar.




(a) Compile-Time Points To Graph (b) Run-Time Pool Descriptor Graph

**Figure 12: Dynamic expansion example**

pool descriptor ($PD$), it is passed additional pool descriptor arguments ($PD_1 \ldots PD_n$) for all the pools to which it points. It adds $PD$ to the "points-from" list of each descriptor $PD_1 \ldots PD_n$. For the example, when the `list2` pool descriptor is initialized, it is passed pointers to the `int` pool descriptor and itself (since the `list2` node has a self-loop), so it adds itself to the points-from lists in both pools. `pooldestroy_pc`($PD$) removes the $PD$ entry from $PD_1 \ldots PD_n$. The run-time points-from lists are created and emptied in this manner because, if $N_1 \rightarrow N_2$ in the compiler's points-to graph, then the lifetime of $P_1$ (for $N_1$) is properly nested within the lifetime of $P_2$ (for $N_2$).

At run time, if the $2^k th$ node is allocated from a pool, $P$, the "points-from" list in $P$ is traversed, decompressing all the pointers in each pool in the list. For example, in Figure 12, when the $2^k th$ node is allocated from the "`list2`" pool, both the `list2` pool and the `list1` pools need to be decompressed so that all pointers into the `list2` pool are $n$-bit values. The normal metadata for a pool identifies which objects in the pool are live. *All* pointers in each live object are decompressed (because of our third restriction above). Decompressing each pointer simply means zero-extending it from $k$ to $n$ bytes. Decompression will grow the pool, which may require additional pages to be allocated and the pool base may change. As objects are copied to their new locations, their relative position in the pool is preserved so that all indices into the pool remain valid. This ensures that no data in any other pool or in globals, stack locations or registers must be modified when a pool is decompressed. (Note, however, that in a multithreaded program, decompressing a pool must be performed atomically relative to any accesses to the pool.)

## 4.3 Interprocedural Dynamic Compression

As noted with static pointer compression, the primary challenge in the interprocedural case is that the same points-to graph node may represent pools containing compressed indices or non-compressed indices. This led to the possibility that functions must be cloned in the static case. Because dynamic pointer compression already uses conditional code to distinguish compressed indices from expanded indices, the need for cloning does not arise. (This solution using conditional code can also be used for the static case if cloning is undesirable or is expensive for a particular function.)

For interprocedural dynamic compression to compress indices in a pool, it must check if the pool meets the first criterion (TH pool, no field address taken) for all calling contexts. Our pointer analysis algorithm computes two DS graphs for each function - a bottom-up (BU) graph representing a function and its callees (but not any callers), and a final, top-down (TD) graph representing the effects of both callees and callers. Therefore, we can check the criterion for all contexts trivially simply by checking it in the TD graph.

| Original Statement | | Transformed Statement |
|---|---|---|
| poolinit(PD, Size) | ⇒ | poolinit_pc(PD, NULL) |
| pooldestroy(PD) | ⇒ | pooldestroy_pc(PD) |

**Figure 13: Rewrite rules for non-compressed pools**

Interprocedural dynamic pointer compression is very straightforward: a single linear pass over the program is used to rewrite all of the instructions in the whole program, according to the rewrite rules in Figure 11 and Figure 13. The only difference between compressed and non-compressed pools (i.e., those that pass or fail the above criterion) is that the `poolinit_pc` call for the latter pool passes a null type descriptor (and an empty points-to list). In this case, `poolinit_pc` initializes the pool descriptor such that `PoolBase` is null and `isComp` is false, and the run-time ensures that the `poolalloc_pc/free_pc` calls behave the same as `poolalloc/poolfree`.

This approach takes advantage of the fact (noted in Section 3.2) that pointers into pools created by the pool allocator do not escape from the program. Because `isComp` is false, non-compressed pools will always use the "expanded" code paths, which use the uncompressed sizes and field offsets for memory accesses.

# 5. OPTIMIZING PTR COMPRESSED CODE

The straight-forward pointer compression implementations described in Sections 3 and 4 generates functional, but slow, code. We describe several straightforward improvements below that can significantly reduce redundant or inefficent operations in the generated code. The first two apply to both static and dynamic compression, while the third applies whenever conditional branches on `isComp` are used.

## 5.1 Address Space Reservation

One of the biggest potential overheads of pointer compression is the need to keep the memory pools contiguous for indexed pools. In particular, any pool must be able to grow in size as memory is allocated from it. (Note that this is unrelated to decompression, and applies to all versions – basic pool allocation, static compression as well dynamic compression.) If the pool allocator is built on top of a general memory allocator like `malloc`, keeping a pool contiguous when growing it may require copying all its data to a new location with enough memory. Although indices into the pool do not have to be rewritten because byte offsets do not change, the data copy can be quite expensive.

Given that this work targets 64-bit address space machines, however, a reasonable implementation approach is to choose a large static limit for individual data structures in the program that is unlikely to be exceeded (e.g., $2^{40}$B), and reserve that much address space for each pool when it is created by the program (using facilities like `mmap(MAP_NORESERVE)`). This allows the program to grow a data structure up to that (large) size without ever needing to copy the pool or move live objects within the pool. The operating system kernel allocates memory pages to the data structure on demand, as they are referenced.

This strategy also ensures that the PoolBase never changes, which can make the next optimization more effective.

## 5.2 Reducing Redundant PoolBase Loads

Pointer compression requires loading the `PoolBase` and `isComp` fields from the pool descriptor for each load and store from a pool. Although these loads are likely to hit in the L1 cache, this overhead can dramatically impact tight pointer-chasing loops. Fortunately, almost all of these loads are redundant and can be removed with Partial Redundancy Elimination (or a combination of LICM and GCSE). The only operation that invalidates these fields is an allocation, either from the pool (moving the pool base) or, in the dynamic case, from one of the pools it points to (decompressing pointers in the pool). The DS graphs directly identify which function calls may cause such operations.

Note that if Address Space Reservation is used, the `PoolBase` is never invalidated, making it reasonable to load it once into a register when the pool is initialized or in the prologue of a function if the pool descriptor is passed in as an argument. Figure 14 shows `MakeList_pc32` after simple optimizations on a 64-bit machine (assuming address space reservation is used).

```
static int MakeList_pc32(Pool *PD, int N) {
  char *PoolBase = PD->poolbase;
  int Result = 0;
  for (int i = 0; i != N; ++i) {
    int Node = poolalloc_pc(PD, sizeof(list_pc32));
    char *NodePtr = Poolbase+Node;
    *(int*)(NodePtr+4) = Result;
    *(int*)NodePtr = i+'A';
    Result = Node;   }
  return Result;  }
```

**Figure 14: `MakeList_pc32` after optimization**

## 5.3 Reducing Dynamic `isComp` Comparisons

The generated code for dynamic pointer compression makes heavy use of conditional branches to test whether or not the pool is compressed. To get reasonable performance from the code, several standard techniques can be used. The most important of these is to use loop unswitching on small pointer chasing loops. This, combined with jump threading (merging of identical consecutive conditions) for straight-line code, can eliminate much of the gross inefficiency of the code, at a cost of increased code size. Other reasonable options are to move the "expanded" code to a cold section vs hot section, or use predication (e.g., on IA64).

# 6. EXPERIMENTAL RESULTS

We implemented the static approach to pointer compression in the LLVM Compiler Infrastructure [9], building on our previous implementation of Data Structure Analysis [7] and Automatic Pool Allocation [10]. We reserve 256MB of memory for each indexed pool using `mmap` to avoid reallocating pools and to make redundancy elimination of PoolBase pointers easier (as described in Section 5). To evaluate the performance effect of Pointer Compression, we first look at how it affects a set of pointer-intensive benchmarks, then investigate how the effect of the pointer compression transformation varies across four different 64-bit architectures.

## 6.1 Performance Results

Figure 15 shows the results of using pointer compression on a collection of benchmarks drawn from the Olden [14] and Ptrdist [3] benchmark suites, plus the LLUbench [20] microbenchmark. These results were obtained on an UltraSPARC-IIIi processor with a 64 kB L1 data cache and a 1 MB unified I+D L2 cache.

To evaluate the performance impact of pointer compression, we compiled each program with the LLVM compiler (including the pool allocation or pointer compression), emitted C code, and compiled it with the system GCC compiler. The PA and PC columns show the execution time for each benchmark with Pool Allocation or Pointer Compression turned on, and the PC/PA column is their runtime ratio. The 'NoPA' column shows the runtime for the program compiled with LLVM and using exactly the same sequence of passes as PA, but omitting pool allocation itself. This is included to show that the pool allocated execution time for the program is a very aggressive baseline to compare against. (We also show the PC/NoPA column because we view pointer compression as a "macroscopic" optimization enabled by pool allocation, and this column shows the aggregate benefit of the approach.) Each number is the minimum of three runs of the program, reported in seconds. To measure the effect on memory consumption, we measured the peak size of the heap for both the pool allocated and pointer compressed forms and the ratio of the two, shown in the last three columns of the table.

Considering llubench first, the table shows that pointer compression speeds up this microbenchmark by over 2x compared with pool allocation (and about 3x compared with the original code). This improvement is achieved by dramatically reducing the memory footprint of the program (and the cache footprint, as will be seen in the next section). Memory consumption is reduced almost exactly by a factor of 2, which is the best possible for pointer compression when compressing from 64 to 32-bits. We use llubench to analyze the behavior of pointer compression in more detail across several architectures in the next section.

Four of the other benchmarks (perimeter, treeadd, tsp, and ft) speed up by 6-32% over pool allocation.[5] All four of these programs show substantial reductions in memory footprint, ranging from 1.33x for tsp to 2x for treeadd.

Of the remaining programs, three (bh, bisort, and power) see no benefit from pointer compression, while two others (em3d and ks) exhibit slowdowns. For all of these except bisort, we see that there is little or no reduction in memory consumption. BH, for example, is not type-homogenous, so pointer compression does not compress anything. Power sees a small reduction but has such a small footprint that its main traversals are able to live in the cache, even with 64-bit pointers. In ks, pointer compression shrunk a pointer but the space saved is replaced by structure padding. Bisort shows a 2x reduction in memory footprint, but does not experience any performance benefit.

[5] Comparing these numbers for the `ft` benchmark with our previous experiments on an AMD Athlon system [10] shows some surprising differences. We have verified that these differences are valid. In particular, ft compiled with GCC (or with LLVM) on x86 is far slower than on Sparc, e.g., 64s. vs. 9.8s. for GCC. Pool allocation speeds up ft by a factor of more than 11x on x86 (compared with GCC or LLVM), indicating that the original code for ft has extremely poor cache behavior on x86. This poor behavior is not observed on the Sparc.

| Program | NoPA (s) | PA (s) | PC (s) | PC/PA | PC/NoPA | PeakPA | PeakPC | PeakPC/PeakPA |
|---|---|---|---|---|---|---|---|---|
| bh | 15.32 | 16.19 | 16.17 | .999 | 1.05 | 8MB | 8MB | 1.0 |
| bisort | 27.33 | 24.31 | 24.29 | .999 | .889 | 64MB | 32MB | 0.5 |
| em3d | 27.19 | 27.16 | 30.25 | 1.11 | 1.11 | 47.1MB | 47MB | 1.0 |
| perimeter | 10.13 | 6.63 | 5.38 | .811 | .531 | 299MB | 171MB | 0.57 |
| power | 6.53 | 6.52 | 6.51 | .998 | .997 | 882KB | 816KB | 0.92 |
| treeadd | 72.83 | 52.78 | 35.76 | .678 | .491 | 128MB | 64MB | 0.5 |
| tsp | 18.33 | 16.28 | 15.28 | .939 | .834 | 128MB | 96MB | 0.75 |
| ft | 14.59 | 11.60 | 10.04 | .866 | .688 | 8.75MB | 4.44MB | 0.51 |
| ks | 8.01 | 7.93 | 8.27 | 1.04 | 1.03 | 47.1KB | 47.1KB | 1.0 |
| llubench | 37.42 | 27.89 | 11.87 | .426 | .317 | 4.49MB | 2.24MB | 0.5 |

**Figure 15: Pointer Compression Benchmark Results (all times in seconds)**

An interesting problem exhibited initially by ks was that using a pre-reserved pool address space of 4MB or any larger size slowed down the PC version by over 60% relative to PA, but smaller pool sizes did not. 4MB is the largest page size on Sparc, and we speculated that this is because of cache conflicts between distinct pools, when two pools of the larger size both start at 4MB boundaries. We improved the runtime library to stagger the start locations of distinct pools differently (relative to a 4MB page boundary). This eliminated the problem in `ks`, producing the results in the table, i.e., only a 4% slowdown. It had a negligible affect on the other programs in the table. For now, we simply start the $i^{th}$ pool at an offset equal to $i \times$`sizeof(`$\tau$`)` bytes, although a better solution long-term may to stagger the start location of pools in a more random manner.

## 6.2 Architecture Specific Impact of Pointer Compression

In order to evaluate the effect of pointer compression on different architectures, we chose to use a single benchmark, LLUbench, and a range of input sizes. We chose LLUbench, a linked-list microbenchmark, because its input size can be scaled over a wide range and it is small enough to get working on several platforms without having to port our entire compiler to each system.

Figure 6.1 shows the scaling behavior of llubench on four different systems, compiled in several configurations. For each configuration, we compiled and optimized the program using LLVM, emitted C code, then compiled the resultant code with a standard C compiler (IBM XLC for the SP, GCC for all others). We used 6 configurations for each platform: the original code (Normal), pool allocation only (PoolAlloc), and pointer compression (PtrComp), each compiled in 32-bit mode and in 64-bit mode (except the Linux Itanium system, which lacks 32-bit support).

The heap size used by llubench is a linear function of the number of iterations, but the execution time of the benchmark grows quadratically. To compare performance of different configurations and systems as a function of the heap size, therefore, we show the ratio of total running time to number of list nodes on the $Y$ axis. Ideally this number should stay constant but in practice, it increases with heap size because the processor spends more time stalled for cache misses[6].

Overall, 64-bit pointers have a major performance overhead compared to 32-bit pointers for all systems, when using either the native (Normal) or pool allocator. With a particular pointer size, the Automatic Pool Allocation transformation consistently increases locality over using the standard system allocator, and the effect is particularly pronounced with the default Solaris malloc implementation. (Automatic Pool Allocation improves locality because it reduces working set sizes by packing individual data structures more closely in memory, and because it improves spatial locality for some data structure traversal patterns [10].)

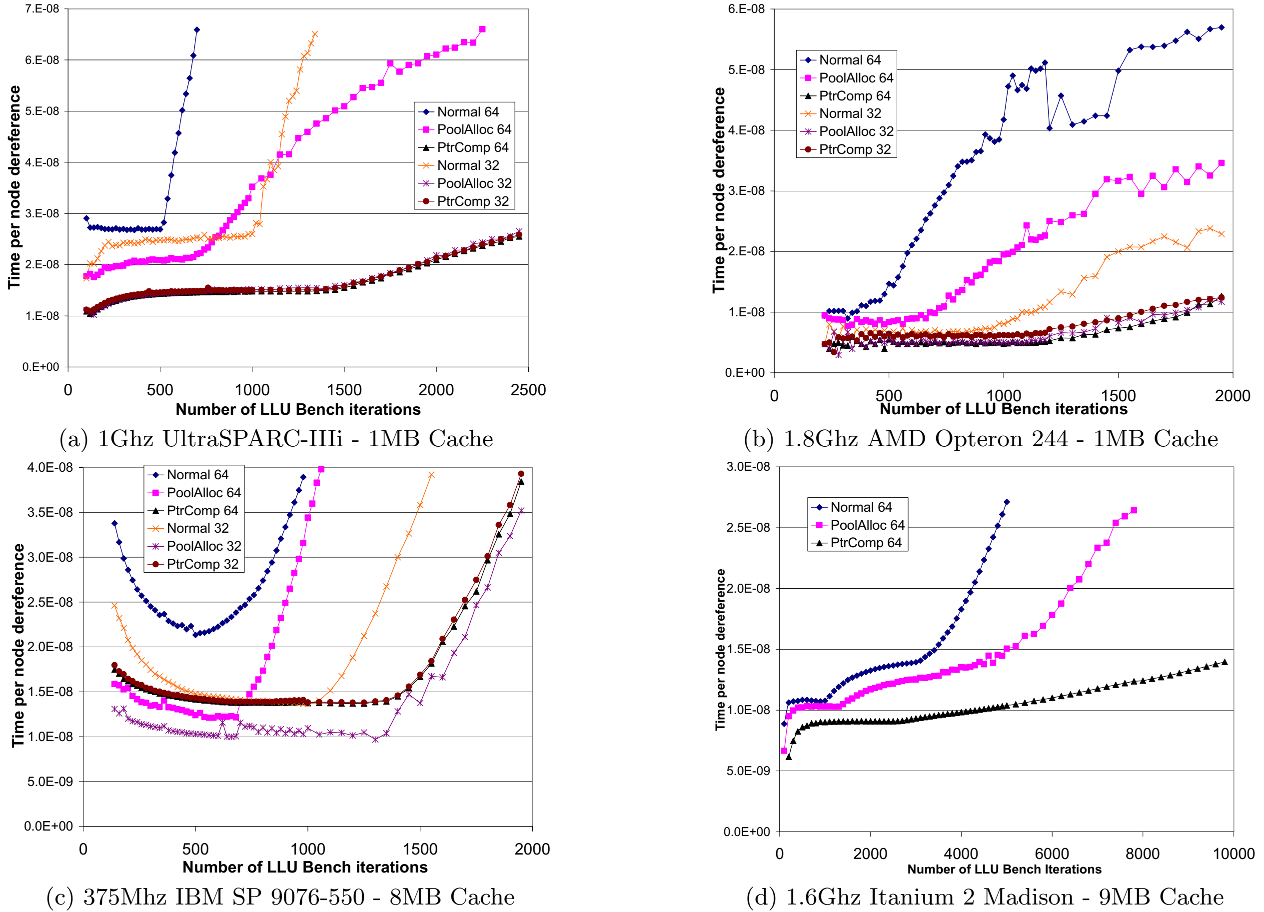
To evaluate the overhead of pointer compression, the "PtrComp 32" values show the effect of transforming 32-bit pointers into 32-bit indexes (i.e. there is no compression, just overhead added). On SPARC, the added ALU overhead of pointer compression is negligible, but on AMD-64 there is a fair amount of overhead because of the extra register pressure (IA-32 has a very small integer register file). On the IBM-SP, pointer compression adds a substantial overhead to the program: the native 64-bit program is faster than the pointer compressed code until about 700 iterations in the program. On this (old) system, the memory hierarchy is fast enough, and the ALUs are slow enough that pointer compression may not make sense.

On the SPARC system, pointer compression provides a substantial speedup over PoolAlloc, and PtrComp-64 is able to *match* the performance of the 32-bit native version. On the Itanium PtrComp makes the code substantially faster across the range of iterations (but we cannot compare to a 32-bit baseline). In the case of the Opteron, PtrComp-64 is actually the fastest configuration (even faster than poolalloc-32): in 64-bit mode the Opteron can use twice as many integer registers as in its 32-bit mode, so it does not need to spill as often. On the IBM SP, performance is substantially improved with pointer compression, but can not match the 32-bit version with pool allocation because of the slow ALU. On all systems though, pointer compression improves the performance of 64-bit applications dramatically as the problem size increases. The figures also show that on all the architectures, the problem size at which performance begins to degrade rapidly is much larger for PtrComp than for PoolAlloc, showing that pointer compression significantly reduces the effective working set size of the benchmark on each of the architectures.

# 7. RELATED WORK

If an architecture supports both 64-bit and 32-bit pointers, and if the application does not require the use of a 64-bit address space, the simplest solution is simply to compile the program in 32-bit mode, which can provide a substantial performance increase [12]. Unfortunately, this approach will not work for many applications that require 64-bit address spaces, e.g., due to genuine use of more than 4GB of memory, due to special requirements for more address space than

[6]Note that the IBM SP system does not support `MAP_NORESERVE`, which significantly increases the time to create a pool (thus impacting runs with a small number of iterations).




(a) 1Ghz UltraSPARC-IIIi - 1MB Cache

(b) 1.8Ghz AMD Opteron 244 - 1MB Cache

(c) 375Mhz IBM SP 9076-550 - 8MB Cache

(d) 1.6Ghz Itanium 2 Madison - 9MB Cache

**Figure 16: llubenchmark: time to process one node vs problem sizes**

physical memory (e.g., [15, 18, 6]), or because the system does not provide 32-bit runtime libraries (e.g. Linux IA-64). Our approach allows an aggregate 64-bit address space and selective compression of individual data structures, where each data structure is limited to 4GB of memory in the static case. In the dynamic case, there is no such inherent limit per data structure.

Most recently, Adl-Tabatabai et. al. describe a trivial form of pointer compression to compile 64-bit pointers in Java programs to a 32-bit pointer model [1]. Their approach is very simple (requiring no program analysis at all), unilaterally compressing pointers to be offsets from the base of the Java memory image located in a 64-bit address space. To decompress these pointers, they add the base of the Java memory image to compressed value, allowing a Java heap size of $2^{32}$ bytes. This approach provides substantial performance improvements, but provides little benefit over having the JVM produce 32-bit code directly.

Ananian and Rinard [2] describe a collection of data size reduction techniques for Java programs, including techniques for arithmetic fields with a limited value range (identified via a bitwidth analysis), unused and constant fields, fields modified only once during initialization, fields that are almost always a single default value, class pointer size reduction, and byte packing for object fields. None of these techniques addresses the size of heap pointers (except the special case of class pointers), and all the techniques are orthogonal to, and could be combined with, our pointer compression approach.

Zhang and Gupta compress pairs consisting of a 32-bit integer and 32-bit pointer into two 15-bit values which are packed into a single 32-bit field [19]. They compress a pointer by replacing it with a value relative to its own address, which is effective for recursive data structures packed closely in memory. If the offset exceeds 15-bits, the pair is replaced with a pointer to an uncompressed pair on the side. They show a substantial reduction in memory consumption, cache misses, and (with custom hardware support) a reasonable performance increase on a subset of the Olden benchmarks. Unlike our work, their transformation is completely manual and only operates on pairs of values (but it can compress integers as well as pointers, and can selectively compress some fields and not others). Also, it requires specialized hardware to improve performance.

Takagi and Hiraki describe a combined hardware/software technique they dub “Field Array Compression Technique” (FACT) [16]. FACT uses manual “Instance Interleaving” [17] to split each structure definition, packing the compressed fields of multiple instances of a structure together in memory. To handle data that cannot be compressed, they always allocate enough space for both the compressed and uncompressed data. This usually improves locality though it does not reduce memory consumption. Compared with our work, FACT has higher memory consumption, requires manual transformation of the program, and requires exotic single-purpose hardware support.

An additional advantage of the macroscopic approach to pointer compression is that it allows standard compiler optimizations (e.g. loop unswitching) to statically optimize the compressed code for specific static pools. In the case of both the Zhang/Gupta and Takagi/Hiraki approaches, the compiler cannot use coarse grain optimizations because individual fields in the heap are compressed or uncompressed unrelated to each other. Using our approach, a compiler can trivially unswitch a dynamic pointer compressed loop that traverses a pool if the loop does not allocate from the pool.

Finally, in an earlier paper [8], we briefly described the idea of using automatic pool allocation for pointer compression, but did not include a specific algorithm nor any implementation or evaluation.

# 8. CONCLUSION

This paper described a sophisticated technique for improving memory consumption and potentially memory system performance on 64-bit targets by shrinking 64-bit pointers to 32-bit values. The transformation builds directly on Automatic Pool Allocation by replacing pointers with pool-relative indices, and then compressing those indices when they are contained within objects used in a type-safe manner (as inferred by an underlying pointer analysis).

Automatic Pool Allocation provides three key benefits for pointer compression. First, it provides a known compile-time mapping between pointers and pools, so that a pool-relative index is sufficient to compute the pointer value. Second, by segregating data structures into distinct pools, it allows $2^{32}$ bytes of address space to be used for each individual data structure. Third, it provides type-homogeneous pools which, in the dynamic case, allows objects to be rewritten safely when a pointer must be expanded in size. Together, these capabilities allow a fairly simple but powerful approach to automatic pointer compression.

The dynamic version of pointer compression is interesting because it could allow indices to start out even smaller than 32 bits (e.g., 16 bits) and grow as needed. We believe that the key challenge in dynamic pointer compression is making accesses to pool objects efficient (and not the dynamic index expansion itself). In fact, the index expansion and data copying are quite simple compared with widely used algorithms for garbage collection today (no pointer traversals are needed, only source pools pointing directly to the indexed pool need to be rewritten, and simple metadata is sufficient to know what fields to expand in each source pool). The key challenge will be making address computations efficient and avoiding conditional code for loading and storing compressible index values.

# Acknowledgements

This work was supported in part by an NSF CAREER Award (EIA-00-93426), the NSF Next Generation Software Program (EIA-01-03756), the MARCO Focus Research Center Program through GSRC, and an Intel Graduate Fellowship. We are grateful to John Criswell for his extensive assistance, both with the implementation of pointer compression and with the experiments. Finally, we thank the anonymous reviewers for their valuable feedback on this work.

# 9. REFERENCES

[1] A. Adl-Tabatabai, J. Bharadwaj, M. Cierniak, M. Eng, J. Fang, B. Lewis, B. Murphy, and J. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. In *Proc. 2004 Int'l Symposium on Code Generation and Optimization*, pages 100–110, Mar. 2004.

[2] C. S. Ananian and M. Rinard. Data Size Optimizations for Java Programs. In *LCTES*, San Diego, CA, Jun 2003.

[3] T. Austin, et al. The Pointer-intensive Benchmark Suite. `www.cs.wisc.edu/~austin/ptr-dist.html`, Sept 1995.

[4] B.-C. Cheng and W. mei Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *PLDI*, pages 57–69, Vancouver, British Columbia, Canada, June 2000.

[5] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *SAS'00: Proc. 7th Int'l Symp. on Static Analysis*, pages 175–198, London, UK, 2000.

[6] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software – Practice and Experience*, 28(9):901–928, 1998.

[7] C. Lattner. *Macroscopic Data Structure Analysis and Optimization.* PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* `http://llvm.cs.uiuc.edu`.

[8] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, Jun 2002.

[9] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. 2004 Int'l Symp. on Code Generation and Optimization (CGO'04)*, San Jose, USA, Mar 2004.

[10] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, Jun 2005.

[11] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analysis. In *SAS*, July 2001.

[12] J. C. Mogul, J. F. Bartlett, R. N. Mayo, and A. Srivastava. Performance implications of multiple pointer sizes. In *USENIX Winter*, pages 187–200, 1995.

[13] E. M. Nystrom, H.-S. Kim, and W. mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *SAS*, 2004.

[14] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *TOPLAS*, 17(2), Mar. 1995.

[15] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. ACM Conf. on Computer And Communications Security*, pages 298–307, 2004.

[16] M. Takagi and K. Hiraki. Field array compression in data caches for dynamically allocated recursive data structure. In *Proceedings of 5th International Symposium on High Performance Computing (ISHPC'03), Tokyo-Odaiba, Japan, October 2003*, pages 127–145.

[17] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 322–329, Oct. 1998.

[18] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *USENIX Summer*, pages 175–186, 1993.

[19] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In *International Conference on Compiler Construction (CC)*, Apr 2002.

[20] C. B. Zilles. Benchmark health considered harmful. *SIGARCH Comput. Archit. News*, 29(3):4–5, 2001.